CDF/xxxx/xxxx/XXX/XXX
version 1.00 April 25, 2003

STNTUPLE manual

R.Culbertson, P.Murat,
Fermi National Accelerator Laboratory

A.Dominguez,
LBNL,

A.Taffard, B.Heinemann,
University of Liverpool

D.Tsybychev,
University of Florida,

J.-Y. Chung, R.Hughes, E.Thomson,
Ohio State University

B.Knuteson,
University of Chicago,

S.Sarkar,

K.Terashi

Abstract

This manual is well known to be incomplete. You're welcome to send us your favorite topics to be included.

1 Introduction

The name "STNTUPLE" stands for 2 related things: a micro-DST CDF data format and a set of utilities to read the data stored in this format. Originally STNTUPLE was designed to be just an extended ntuple format, it still remains such. Format of ROOT files, however, allows to minimize the difference between the ntuple and micro-DST, this is how STNTUPLE became a micro-DST.

2 Getting Started

In this section includes several examples illustrating how to start using STNTUPLE for analysis quickly.

2.1 STNTUPLE executables

Stntuple package builds several executables.

- **stnmaker.exe** is a light-weight executable, which runs on the output of ProductionExe (cdfSim, cdfGen) and produces output in STNTUPLE format. stnmaker.exe doesn't recalculate parameters of the input objects and is stable in time. Use this executable if for the generator-level studies. To build stnmaker.exe do

  ```
  make Stntuple.stnmaker
  ```

  stnmaker.exe is being built and validated nightly.

- **stnmaker_prod.exe** - this purpose of this executable is to generate STNTUPLE's with most up-to-date information. It runs on the output of ProductionExe, remakes high-level objects and stores the information in STNTUPLE files. **stnmaker_prod.exe** also runs the folowing analysis modules, which are not part of Production: **CosmicFinderModule, JetProbModule, TrackRefitterModule, SecVtxModule, TopEventModule**

  Module contents of **stnmaker_prod.exe** may change significantly from one release to another. To build this executable against CDF offline release 4.9.1:

  ```
  setup cdfsoft2 4.9.1
  newrel -t 4.9.1 dev_239
  cd dev_239
  addpkg Stntuple dev_239
  chmod 755 Stntuple/scripts/build_stntuple
  Stntuple/scripts/build_stntuple
  gmake Stntuple._prod
  ```

  The latest instructions on how to build the current version of **stnmaker_prod.exe** can be found at **STNTUPLE home page**

- **stnmakerFat.exe** - specialized executable for B-tag studies. Aaron knows more about its contents.

3 Storing collections

3.1 Naming

- Each collection of CDF objects can be identified by 2 strings: its **description** and **name of the creating process** - see AC++ docs for more details. To reflect this mnemonics, in StntupleMaker talk-to's CDF collection names are defined as **"process_name@description"**, where the process name and the description are separated by the column, for example, **PROD@COT_Global_Tracking**.
- "@" is used instead of ":" to allow working with the corresponding branches in split mode
- it is possible to omit process name and refer to the collection by its description only, for example, **COT_Global_Tracking**. In this case default process name is assumed. The default process name can be redefined in the talk-to to StntupleMaker, for example:

  ```
  talk StntupleMaker
    processName set L3
  exit
  ```

3.2 The Defaults

CDF data model allows to store several collections of the same type in the event record, for example, collections of jets reconstructed by different algorithms. Same holds for tracks. By default StntupleMaker stores only one collection of each type in the STNTUPLE. The defaults are

- the default process name is **PROD**
- defTracks collection coresponding to the default process is **always** stored in the '**TrackBlock** data block.
- **JetCluModule-cone0.4** collection corresponding to the default process is stored in the data block with the name **JetBlock**

3.3 Storing more than one collection

It is possible to store more than one collection of tracks or jets in STNTUPLE. The rules are as follows.

- tracks:
  - the following talk-to

    ```
    talk StntupleMaker
      trackCollName set PROD@COT_Global_Tracking_SL  COT_Global_Tracking_HL
    exit
    ```

adds **COT_Global_Tracking_SL** collection created by the process **PROD** and **COT_Global_Tracking_HL** collection created by default process to the list of stored track collections. If default process name is **PROD** the collections will be stored in the data blocks named **PROD@COT_Global_Tracking_SL** and **PROD@COT_Global_Tracking_HL** correspondingly

- rules for the jets are slightly different. When several jet collections are specified in the talk-to, the first one is stored on the branch with predefined name - **JetBlock**, the rest collections are stored on the branches with the names defined by the names of the collections. For example,
  - the following talk-to

    ```
    talk StntupleMaker
       jetCollName set JetCluModule-cone0.7 PROD@JetCluModule-cone0.4
    exit
    ```

    instructs StntupleMaker to store jet collection **JetCluModule-cone0.7** in the data block with the name **JetBlock** and collection **JetCluModule-cone0.7** created by the process **PROD** in the data block named **PROD@JetCluModule-cone0.4**.

Therefore the difference between the tracks and the jets is that for the tracks the contents of the default track block (**TrackBlock**) is predefined, while for the jets there is no such a limitation. The reason is that **defTracks** track collection is used by the electron, muon and tau reconstruction algorithms, so storing it in the STNTUPLE on a branch with predefined name simplifies the analysis code.

4 STNTUPLE and DB Constants

Some run-dependent constants (luminosities, beam positions etc) are usually needed even at the latest stages of the analysis. Such constants are stored (once per run) in the STNTUPLE files and retrieved as necessary. The constants can be accesses via STNTUPLE database manager (class TStnDBManager). All the STNTUPLE "DB tables" are named and accessed by name. Tables stored in the STNTUPLE along with the their respective names are listed below.

- trigger tables, table name "TriggerTable", class TStnTriggerTable
- COT beam positions, table name "CotBeamPos", class TStnBeamPosition.
- SVX beam positions, table name "SvxBeamPos", class TStnBeamPosition.
- run summary constants, table name "RunSummary", class TStnRunSummary.

The following simple example illustrates how to access the DB constants stored in STNTUPLE.

```
//_____________________________________________________________________________
int TTrigAnaModule::BeginRun() {
                                        // talk to DB manager and retrieve the
                                        // trigger table for this run

  TStnDBManager*   dbm = TStnDBManager::Instance();

  TStnTriggerTable* tt = (TStnTriggerTable*) dbm->GetTable("TriggerTable");
....

                                        // get COT beam position at Z=0

  TStnBeamPos* bp = (TStnBeamPos*) dbm->GetTable(``CotBeamPos'');
  Double_t x0     = bp->X0();
  Double_t y0     = bp->Y0();
}
```

5 Advanced topics

5.1 Luminosity information

- Instantaneous luminosity for an event is stored in the header block (see TStnHeaderBlock). It is accessible from within any analysis module as follows:

  ```
  float inst_lum = GetHeaderBlock()->InstLum()
  ```

  luminosity is stored in units of $cm^2 \cdot sec^{-1}$, so the numbers are of the order of 10^{30}

- When stntupling job runs, TStnDBManager updates the luminosity info once per begin run record, however, db record is saved into the ntuple file only once. If events from the same run end up in 2 different files, each of the files has a db subdirectory for this run with the luminosity information.

  STNTUPLE analysis job creates a list of processed runs which it updates as the execution progresses. This list doesn't have duplicates and each run appears in this list only once. As such this procedure doesn't involve any double counting and the only assumption made is that for each run we are processing all its run sections (which is almost always true modulo bad/lost files). The right way is to integrate luminosity in the stntupling job by counting each processed run section separately - this is not implemented yet, volunteers welcome and I can even show how to do it. There are also 3 histograms created by default by the analysis job - they contain delivered luminosity, online luminosity and offline luminosity for the runs which have at least one event stored in STNTUPLE (this is to be fixed)

5.2 Track extrapolation to CES and CMU

STNTUPLE has simple utilities for track extrapolation. To use them one needs to build (standalone) library **libStntuple_geom.so** and load it into the ROOT session:

```
make Stntuple._geom
root
root [0] .L ./shlib/\$BFARCH/libStntuple_geom.so
```

- Classes compiled into this library are defined in Stntuple/geom subpackage
- TSimpleExtrapolator class has methods to extrapolate track up to CES and CMU. Example of its use is shown below:

```
//-------------------------------------------------------------------------------
//  calculate parameters of the SEED track at CES and do it once per tau
//-------------------------------------------------------------------------------
    TSimpleExtrapolator* fExtrapolator     = new TSimpleExtrapolator();
    TTrajectoryPoint   p0;
    TStnTrk* trk = ...;                // defined elsewhere
    TLorentzVector* tmom;

    double xyz[8], xw, zw, ptot;
    int    trk_charge, side, wedge;

    xyz[0]  = -trk->D0()*sin(trk->Phi0());
    xyz[1]  =  trk->D0()*cos(trk->Phi0());
    xyz[2]  =  trk->Z0();

    tmom = trk->Momentum();

    ptot    = tmom->P();

    xyz[3]  = tmom->Px()/ptot;
    xyz[4]  = tmom->Py()/ptot;
    xyz[5]  = tmom->Pz()/ptot;

    xyz[6]  = 0;
    xyz[7]  = ptot;

    p0.SetPoint(xyz);
    trk_charge  = trk->Charge();
    fExtrapolator->SwimToCes(&p0,trk_charge,side,wedge,xw,zw);

                                  // track coordinates at CES
    float track_x_ces = xw;
    float track_z_ces = zw;
-------------------------------
```

5.3 Remote Data Access

To access the STNTUPLE's stored on different platform over the network one can use ROOTD daemon. For the CDF (read-only) installation of ROOTD one can do it as follows:

```
root[0] TStnAna x(‘‘root://fcdfsgi2.fnal.gov//cdf/data05/my_file.stn’’);
```

Use TFile::Open to open network files interactively from the ROOT prompt.

5.4 Copying and filtering

- TStnAna job can do filtering and write output STNTUPLE with the events of interest. Output mode is turned off be default. To turn output on one needs to define an output module, as shown in the example below.
- Every TStnModule can be used as a filter.

```
TStnAna* x;

int test_output() {
  x = new TStnAna("input.stntuple");

  m_l3t = new TTauL3TriggerModule;
  x->AddModule(m_l3t);
  m_l3t->SetFilteringMode(1);   // 0: disabled (default), 1: filter, 2: veto

  // use TStnModule::SetPassed(Int_t Passed) inside the module
  // to

  TStnOutputModule* m = new TStnOutputModule("output.root");
  x->SetOutputModule(m);
  x->Run(100);
}
```

- Primary STNTUPLE (created by the AC++ job) contains DB records, including the luminosity information, for all the runs seen by InitStntupleModule. Secondary STNTUPLE's will contain DB information only for those runs, for which it at least one event has been written into the ntuple.

6 Identification of the High-Level Objects

Classes TStnTauID, TStnElectronID etc provide utilities for identification of the high-level physics objects. We consider typical use case of these classes taking TStnTauID class as an example:

```
  TStnTauID* fTauID = new TStnTauID();
  int ntau = fTauBlock->NTaus();
  for (int i=0; i<ntau; i++) {
    TStnTau* tau = fTauBlock->Tau(i);
    int id_word = fTauID->IDWord(tau);

    if (id_word == 0) {
                              // the tau passed all the ID cuts
     ..........
    }
  }
```

TStnTauID::IDWord checks the tau identification cuts for a given tau candidate and returned integer - **id_word** - is a bit-packed mask defining passed and failed cuts. Each ID cut implemented in TStnTauID has a bit number assigned to it. When a cut fails, the bit corresponding to it is set to 1. **id_word=0** means that tau object passed all the selection cuts, the next example shows how to check whether a given tau object passed given ID cut:

```
 TStnTauID* id = fTauID;
...
 int id_word = fTauID->IDWord(tau);
 if ((id_word & ~ id->kCalIsoBit) == 0)  {
                                       // tau candidate passed calorimetry
                                       // isolation cut
  .............
 }
```

7 TStnTauID

TStnTauID class provides set of tau identification utilities. Two most important methods of the class are TStnTauID::IDWord and TStnTauID::LooseIDWord, which implement “tight” and “loose” definitions of tau leptons.

7.1 Bits and ID cuts

Assignment of the bits is described in TStnTauID.hh include file.

```
kDetEtaBit              = 0x1 <<  0,
kEtBit                  = 0x1 <<  1,
kSeedTowerEtBit         = 0x1 <<  2,
kSeedTrackPtBit         = 0x1 <<  3,
kEmfrBit                = 0x1 <<  4,
kNTrk1030Bit            = 0x1 <<  5,
kNPi01030Bit            = 0x1 <<  6,
kCalIsoBit              = 0x1 <<  7,
kCalIso1Bit             = 0x1 <<  8,
kTrkIsoBit              = 0x1 <<  9,
kNAxSegBit              = 0x1 << 11,
kNStSegBit              = 0x1 << 12,
kSeedTrackD0Bit         = 0x1 << 13,
kSeedTrackZ0Bit         = 0x1 << 14,
kSeedTrackDzBit         = 0x1 << 15,
kZCesBit                = 0x1 << 16,
kCalMassBit             = 0x1 << 17,
kTrkMassBit             = 0x1 << 18,
kVisMassBit             = 0x1 << 19,
kNMuStubsBit            = 0x1 << 20, // this shouldn't really be used
kTightElectronBit       = 0x1 << 21,
kTrkAngleBit            = 0x1 << 22,
kPi0AngleBit            = 0x1 << 23
```

7.2 Int_t TStnTauID::IDWord(TStnTau* Tau)

Implements the following ID cuts:

- $| \eta_{\rm detector} | < \eta_{\max}$ (default = 1.0)
- $\rm E_t\ > E_{t\ min}$ (default = 20.0)
- $\rm E_{t\ seed\ tower} > E_{t\ min}$ (default = 10.0)
- $\rm P_{t\ seed\ track} > P_{t\ min}$ (default = 4.5)
- electron rejection cut. Depending of the value of TStnTauID::fSelectElectrons this cut can be reversed.
  - TStnTauID::fSelectElectrons = 0 selects electron rejection mode:

$$\rm EM_{fraction}(\tau)\ <\ 1-\xi/(E/p)$$

  - TStnTauID::fSelectElectrons = 1 inverts the cut to select electrons:

$$\rm EM_{fraction}(\tau)\ >\ 1-\xi/(E/p)$$

- $\rm N_{tracks}(10-30)\ <\ N_{max}$ (default : $\rm N_{max}\ =\ 1$)

7.3 Int_t TStnTauID::LooseIDWord(TStnTau* Tau)

"Loose" tau identification cuts are a subset of the "tight" set of cuts:

- $| \eta_{\rm detector} | < \eta_{\max}$ (default = 1.0)
- $\rm E_t\ > E_{t\ min}$ (default = 20.0)
- $\rm E_{t\ seed\ tower} > E_{t\ min}$ (default = 10.0)
- $\rm P_{t\ seed\ track} > P_{t\ min}$ (default = 4.5)
- electron rejection cut. Depending of the value of TStnTauID::fSelectElectrons this cut can be reversed.
  - TStnTauID::fSelectElectrons = 0 selects electron rejection mode:

$$\rm EM_{fraction}(\tau)\ <\ 1-\xi/(E/p)$$

  - TStnTauID::fSelectElectrons = 1 inverts the cut to select electrons:

$$\rm EM_{fraction}(\tau)\ >\ 1-\xi/(E/p)$$

8 STNTUPLE Datasets

In this section we consider structure and cataloging of the STNTUPLE static datasets and describe the cataloging procedure. Static STNTUPLE datasets can be distributed over the network and span over multiple static fileservers. Their structure is very similar to the structure of the datasets described in the CDF Data File Catalog. STNTUPLE data catalog, however, is implemented as a text database, maintenance of which doesn't require any specialized tools.

8.1 Format of the STNTUPLE Dataset Catalog

STNTUPLE data catalog is located in the directory pointed to by the **STNTUPLE_CATALOG** environment variable. The following example defines STNTUPLE data catalog to be located in the $HOME/catalog subdirectory (we assume **bash** shell is being used):

```
 export STNTUPLE_CATALOG=\$HOME/catalog
```

Datasets can de versioned. For example, one can think of creating STNTUPLE's using bf stnmaker_prod.exe version dev_239 and comparing them to the datasets created with the previous version of stnmaker_prod.exe (version dev_238). STNTUPLE catalog therefore may have multiple partitions, and we use the same notation as CDF DFC - **book** - for those. A **book** is a subdirectory in the STNTUPLE catalog directory, name of the book is the relative name of the subdirectory.

For example, STNTUPLE data catalog described by the following directory structure

```
/data12/murat/tgeant/junk:
used 8 available 1628008
drwxr-xr-x    2 murat    cdf          4096 Apr 23 21:52 .
drwxr-xr-x   26 murat    cdf          4096 Apr 23 21:52 ..
drwxr-xr-x    2 murat    cdf          4096 Apr 23 21:52 murat
drwxr-xr-x    2 murat    cdf          4096 Apr 23 21:52 stntuple

/data12/murat/tgeant/junk/murat:
used 16 available 1627984
drwxr-xr-x    4 murat    cdf          4096 Apr 23 21:52 .
drwxr-xr-x    4 murat    cdf          4096 Apr 23 21:53 ..
drwxr-xr-x    2 murat    cdf          4096 Apr 23 21:52 test1
drwxr-xr-x    2 murat    cdf          4096 Apr 23 21:52 test2

/data12/murat/tgeant/junk/stntuple:
used 16 available 1627984
drwxr-xr-x    4 murat    cdf          4096 Apr 23 21:53 .
drwxr-xr-x    4 murat    cdf          4096 Apr 23 21:53 ..
drwxr-xr-x    2 murat    cdf          4096 Apr 23 21:53 dev_238
drwxr-xr-x    2 murat    cdf          4096 Apr 23 21:53 dev_239
```

has 4 books: **stntuple/dev_238, stntuple/dev_239, murat/test1** and **murat/test2** defined. The book names should not contain ":" character in it.

A book may have several datasets described in it, there is no limitations on the syntax of the dataset names except the one above. All the names above should be valid UNIX directory names. A full dataset name is specified by the book and by the dataset ID separated by ":", for example:**stntuple/dev_239:btop0g** or **murat/test1:zee_new**.

A dataset consists of several filesets, a fileset is a set of files located on the same fileserver in the same directory. Different filesets of the same dataset can reside on different fileservers. There is no limitations on the fileset name or number of files in one fileset, other than discussed above. Dataset catalog resides in the directory, which name is defined by the names of the book and the dataset, for example, catalog of the dataset **stntuple/dev_239:btop0g** is located in $STNTUPLE_CATALOG/stntuple/dev_239/btop0g.

8.2 Catalog Files

Dataset catalog consists of several ASCII files residing in the same directory. If dataset has N filesets in it, the dataset catalog consists onf N+1 ASCII files:

- dataset catalog: one file per dataset. It contains fileset-level description of the dataset, its name is fixed by the dataset ID. Fileset-level catalog of dataset ID **btop0g** is stored in the file named **.btop08.catalog** . The following example explains format of the dataset catalog file:

```
#------------------------------------------------------------------------
# fcdfdata051: /cdf/scratch/cdfdata/scratch/btop0g
# fileset     server name             subdirectory
#------------------------------------------------------------------------
 GI0741     fcdfdata051.fnal.gov  /cdf/scratch/cdfdata/scratch/btop0g
 GI0744     fcdfdata051.fnal.gov  /cdf/scratch/cdfdata/scratch/btop0g
 GI0747     fcdfdata051.fnal.gov  /cdf/scratch/cdfdata/scratch/btop0g
 GI0749     fcdfdata051.fnal.gov  /cdf/scratch/cdfdata/scratch/btop0g
 GI0750     fcdfdata051.fnal.gov  /cdf/scratch/cdfdata/scratch/btop0g
 GI0776     fcdfdata051.fnal.gov  /cdf/scratch/cdfdata/scratch/btop0g
 GI0943     fcdfdata051.fnal.gov  /cdf/scratch/cdfdata/scratch/btop0g
 GI0945     fcdfdata051.fnal.gov  /cdf/scratch/cdfdata/scratch/btop0g
 GI0949     fcdfdata051.fnal.gov  /cdf/scratch/cdfdata/scratch/btop0g
 GI1239     fcdfdata051.fnal.gov  /cdf/scratch/cdfdata/scratch/btop0g
 GI1241     fcdfdata051.fnal.gov  /cdf/scratch/cdfdata/scratch/btop0g
 GI1275     fcdfdata051.fnal.gov  /cdf/scratch/cdfdata/scratch/btop0g
 #------------------------------------------------------------------------
 # end of the fileset-level catalog
 #------------------------------------------------------------------------
```

  The dataset catalog has 1 line per fileset, the first column defines the name of the fileset, the 2nd column defined the name of the remote server (may be the name of the local host) and the last column defines the name of the directory where the fileset is located. Lines starting from "#" are the comment lines.

- Catalogs of individual filesets - 1 file per fileset. The name of the file shoudl be the same as the name of the fileset. Fileset catalog includes one non-comment line per file and its format is illustrated by the example below:

```
#-----------------------------------------------------------------------
# GI0741.*
#-----------------------------------------------------------------------
GI0741.0 btop0g.0016.GI0741.0.s.0001 1212517997
GI0741.1 btop0g.0017.GI0741.1.s.0001 1382061686
GI0741.2 btop0g.0018.GI0741.2.s.0001 1170985582
GI0741.3 btop0g.0019.GI0741.3.s.0001 1180158114
GI0741.4 btop0g.0020.GI0741.4.s.0001 1127961129
GI0741.5 btop0g.0021.GI0741.5.s.0001 1074892152
#-----------------------------------------------------------------------
# end
#-----------------------------------------------------------------------
```

The 1-st column is a string which includes the fileset name, the 2-nd column is the file name, the 3rd column is the file size (not used for the moment).

8.3 Using cataloged datasets

The following example explains use of the cataloged dataset.

```
------------------------------------------------ tau_ana.C
 TStnAna* x = NULL;
 TChain* chain = NULL;
 TStnCatalog* catalog = NULL;

 void tau_ana(const char* Dataset, Int_t NEvents = 0) {

   if (! chain) {
     chain = new TChain("STNTUPLE");
     catalog = new TStnCatalog();
     catalog->InitChain(chain,Dataset);
 //
 // example: TStnCatalog::InitChain(chain,"stntuple/dev_239:gqcd1g")

     x = new TStnAna(chain);
   }
 ...
   x->Run(NEvents);
 }
--------------------------------------------- end of tau_ana.C
....
```

The analysis driver script **tau_ana.C** can be called from the ROOT prompt as follows:

- to process the whole dataset:

```
    root [0] .x tau_ana.C(‘‘stntuple/dev_239:btop0g’’);
```

- to process a single fileset:

```
root [0] .x tau_ana.C(‘‘stntuple/dev_239:btop0g:GI0741.0’’);
```

- to process a single file:

```
root [0] .x tau_ana.C(‘‘stntuple/dev_239:btop0g:GI0741.0:btop0g.0016.GI0741.0.s.000
```

- to process first 100 events from the fileset:

```
root [0] .x tau_ana.C(‘‘stntuple/dev_239:btop0g:GI0741.0’’,100);
```

9 Using Good Run Lists - quick instructions

- by default all runs are good
- use TStnAna::SetGoodRunList("ETF") to specify ETF good run list
- use TStnAna::SetGoodRunRoutine to specify your own definition of a good run. Parameter in TStnAna::SetGoodRunRoutine call is a function, returning 1 for good runs and 0 for bad ones. Example:

```
  int function my_good_run(int RunNumber) {
    if (RunNumber < 154799) return 1;
    else                    return 0;
  }

  TStnAna  x;
  x->SetGoodRunRoutine(my_good_run);
```

- here is a brief example of TStnGoodRunList usage:

```
 TStnGoodRunList grl;
 TStnRunSummary* rs;

 grl.Init();

 int run_number = 138425;

 rs = grl.GetRunSummary(run_number);
 float lumi = rs->OfflineLumiRS();

// you can also do

 rs->Print();
```

- technical note: TStnGoodRunList::GetRunSummary returns a pointer to an internally cached structure, so run summary information is available for only one run at a time

9.1 More about TStnGoodRunList

```
TStnGoodRunList grl;
TStnRunSummary* rs;

grl.Init();
```

10 MC objects in STNTUPLE

MC information can be represented at 2 different levels - output of the MC event generators and output of the detector simulation. Correspondingly, STNTUPLE includes 2 different MC blocks:

- TGenpBlock, which contains generator-level information
- TObspBlock, containing data corresponding to the output fo detector simulation

10.1 TGenpBlock

10.2 TObspBlock

10.3 Looping over GENP particles

- The following example shows how to loop over the particles stored in GENP block

```
for (int i=0; i<fGenpBlock->NParticles(); i++) {
  TGenParticle* p   = fGenpBlock->Particle(i);
  int im            = p->GetFirstMother();
  if (im >= 0) {
    TGenParticle* mom = fGenpBlock->Particle(im);
    if (mom!=0) {
      int mom_id = mom->GetPdgCode();
                                    // do something with it
      ....
    }
  }
}
```

Note, that for incoming particles the following line

```
int im = p->GetFirstMother();
```

sets **im** to -1, which means that these particles do not have a mother. Therefore it is necessary to check that **im** is not equal to -1.

11 STNTUPLE Validation tools

This section describes STNTUPLE validation tools. Two most typical validation tasks are

- **code validation:** a data file has been processed with some old "reference" version of the offline code. The same data file has been reprocessed with the newer version of the reconstruction code. How is it possible to find and quantify the differences between 2 different versions of the offline code?
- **data validation:** 2 different data files corresponding to the runs taken under different conditions have been processed with the same version of the reconstruction code. How it is possible to find and quantify the differences between the data?

Steps:

- run 2 STNTUPLE jobs on 2 files in question and for each job save resulting histograms into the coresponding histogram file using TStnAna::SaveHist. Suppose the names of the 2 files are **old.root** and **new.root** correspondingly.
- build validation library **libStntuple_val.so**:

```
make Stntuple._val
```

- start interactive ROOT session, load **libStntuple_val.so**, then do:

```
root[0] .L shlib/$BFARCH/libStntuple_val.so
root[1] compare_stn_hist("old.root","new.root",min_prob)
```

  where **min_prob** is the probability of KS test below which 2 histograms will be considered different. In the end **compare_stn_hist** pops up a TBrowser window and displays histograms with comparison of which has KS probability below **min_prob**

- go to the "root" folder (double click on it with the left button) using left mouse button
- double click on **STNTUPLE_RESULTS**. Icons with the red dot mark modules for which 2 files have different histograms continue clicking with the left button, until you get the histogram names displayed.
- click on a histogram with the right button, you get context menu
- click on "DrawEP", see histograms from the 2 files overlayed on top of each other. In the shell window where you started ROOT, you'll get number of entries and the probability of KS test printed.

12 STNTUPLE Data Blocks

This section contains description of STNTUPLE data structures (blocks)

12.1 TStnElectronBlock

TStnElectronBlock stores information about the reconstructed electron candidates.

12.2 TStnMuonBlock

TStnMuonBlock stores information about the reconstructed muon candidates.

12.3 TStnMetBlock

TStnMetBlock stores variables related to the calculation of the missing Et. As there are different ways to calculate met, TStnMetBlock includes several values of MET, calculated under different assumptions.

- TStnMetBlock::Met(0): MET calculated at Z=0
- TStnMetBlock::Met(1): MET calculated at best VXPRIM vertex (best vertex in "VertexCollection". If an event contains a high-Pt lepton (electron, muon or tau-candidate), Met is calculated at Z0 of the lepton's track
- TStnMetBlock::Met(2): MET calculated at Z event with
- TStnMetBlock::Met(3): MET calculated at best Beate's vertex
- TStnMetBlock::MetPhi(i): Phi angle of the 2-D vector of $\not{E}_{\mathrm{T}}$, corresponding to i-th way of calculating $\not{E}_{\mathrm{T}}$.
- TStnMetBlock::MetX(i): X-component of the 2-D vector of $\not{E}_{\mathrm{T}}$, corresponding to i-th way of calculating $\not{E}_{\mathrm{T}}$.
- TStnMetBlock::MetY(i): Y-component of the 2-D vector of $\not{E}_{\mathrm{T}}$, corresponding to i-th way of calculating $\not{E}_{\mathrm{T}}$.
- TStnMetBlock::Sumet(0): Sum Et over all the calorimeter towers (currently at Z=0)
- TStnMetBlock::Sumet(1): Sum Et calculated at Z of the first vertex in VertexBlock or at Z0 of the high-Pt lepton
- TStnMetBlock::Sumet(2): Sum Et calculated at Z of the highest-Pt vertex in ZVertexBlock
- TStnMetBlock::MetSig(): significance of MET (so far calculated at Z=0)

12.4 TObspBlock

TObspBlock (OBSP - name, which roots are in CDF Run I history) stores post-simulation information: MC particles and vertices produced by the detector simulation code. For the same historical reasons TObspBlock stores FORTRAN-type indices (starting from 1).

Example 1: finding MC vertex, coresponding to track Trk, reconstructed in MC event

```
//_______________________________________________________________________________
int find_mc_vertex(TStnTrack* Trk) {
  // note iv-1 in the vertex index calculation

  int iobsp = Trk->ObspNumber();

  TObspParticle* p;
  TObsvVertex*   v = NULL;

  if (iobsp >= 0) {
    p   = fObspBlock->Particle(iobsp);
    iv  = p->VertexNumber();
    v   = fObspBlock->Vertex(iv-1);
  }
  return v;
}
```

12.5 TStnTauBlock

TStnTauBlock stores information about the reconstructed tau lepton candidates.

12.6 TStnTrackBlock

TStnTrackBlock stores tracking information.

Matching between the tracks and the MC particles is done based on hit parentage using output of TrackObspMatch module - CdfTrackMatch object is used. If TrackObspModule has not been run upstream, matching is not done. Note, that TrackObspModule requires track hits to be present in the event record, so it either has to be run in Production, or Produciton tcl files should be modified to save COT hits - by default they are not saved.

13 Silicon Blocks

There are several different branches with information related to the silicon detector and tracks with silicon hits on them. As in other `STNTUPLE` branches, the data is organized as an object which inherits from TObject and which contains as data members a **TClonesArray** of objects which describe the fundamental unit of interest, such as a silicon cluster, and possible links (see section on `TStnLinkBlock`) to other branches.

The following sections describe the ntuple blocks that contain information on the position of all the wafers, strips, clusters, intersections of tracks with wafers, intersections of `OBSP` particles with wafers, and links between the tracks and silicon hits and intersections. These branches contain enough information to do cluster-level studies, intrinsic resolution studies, and track based studies. Examples of these can be found in the `Stntuple/ana` subdirectory: **TSiPed**, **TRPhiWt**, and **TSiExample**.

First, the fundamental objects used in the `TClonesArray` are described, and then how they fit into the larger block which is stored in a single branch.

13.1 `TStnSiDigiCode`

This class is used to give every readout unit of the silicon detector a unique identifier. It is based on the CDF class `TrackingObjects/SiData/SiDigiCode`. It is used in most of the higher level silicon data blocks in `STNTUPLE` to distinguish data coming from individual half-ladders.

- Data members
  - `UShort_t fDigiCode`: Unique code for each readout unit (also called a half-ladder). This number is fairly compact, and ranges from 0 for L00, barrel 0, phi wedge 0, to 9135 for ISL layer 7, barrel 2, phi wedge 35. Since it is fairly compact, it can be used directly as an index for arrays.
- Member functions
  - `UInt_t Barrel()`: The barrel of this half-ladder. Ranges from 0-2.
  - `UInt_t LadderSeg()`: The ladder segment. Ranges from 0-2 for L00, and 0-1 for SVX and ISL.
  - `UInt_t PhiWedge()`: Phi wedge. Ranges from 0-12 for L00 and SVX, 0-24 for ISL central barrel 1, 0-28 for ISL forward/backward barrels layer 6, and 0-35 for ISL layer 7.
  - `UInt_t Layer()`: Layer number. 0 for L00, 1-5 for SVX, 6-7 for ISL.
  - `UInt_t Side()`: Axial or stereo side of readout unit. 0 for axial, 1 for $90°$ or shallow stereo.
  - Setters: There are also setter functions for all the above quantites.

13.2 `TSiAlign`

This is a simple container `TVector3`'s which represents the center and normal vector of each silicon ladder segment (see Section 13.1) and each wafer in the half ladder.

The input files come from running `TrackingUserMods/test/siGeometryValidation.cc` with the option "outputFile" to save an ascii file of the geometry after alignment. Two example ascii files are in the CVS repository[1].

You can create this object either from an ascii file using the constructor that takes a filename as input, or you can stream it in and out directly since it inherits from `TObject`.

This object defines a simple nested class, `TSiHalfLadder`, which contains the vectors for each wafer.

- Nested class `TSiHalfLadder`
  - `TVector3 h`: Global coordinate (cm) of center of each half ladder
  - `TVector3 nh`: Normal vector of center of each half ladder
  - `int n`: Number of wafers in this half ladder. L00 and SVX half ladders each have two wafers of silicon, whereas ISL half ladders have three.
  - `TVector3 w[3]`: Global position of center of each wafer in the half ladder.
  - `TVector3 nw[3]`: Normal vector of center of each wafer in the half ladder.
- Data members
  - `TMap fMap`: A map of `TSiHalfLadder`'s used as a simple lookup table for the wafer positions.
- Member functions
  - `TVector3* GetCenter(TStnSiDigiCode*)` and `GetNormal(TStnSiDigiCode*)`: Return the global coordinates and normal vector of the center of the half ladder for a given digicode.
  - `int GetNWafer(TStnSiDigiCode*)`: The number of wafers in this half ladder
  - `TVector3* GetWaferCenter(TStnSiDigiCode *digi, int iw)` and `GetWaferNormal(TStnSiDi` `*digi, int iw)`: Return the center and normal vector of a given wafer for this half ladder.

13.3 TStnSiStrip

This class contains all the relevant information about an individual strip in the silicon detector. It is used in the ntuple branch `TSiStripBlock` (see Section 13.4).

Information about the strip is packed into integers and chars to save space, but is puffed up into human readable form by the streamer. Because of this feature, a `TStnSiStrip` must never be split (use split mode -1) since ROOT's default streamer will not unpack the information. What is described below are the human readable data members which are not persistent.

- Data members
  - `Int_t fStrip`: The strip number. The range of this variable depends on which layer and side it comes from, but the maximum allowed range is 0-895.

[1]See `Stntuple/db/si/siGeo_100030_1_GOOD.txt` for the ascii file that represents the silicon alignment used for the winter conferences of 2003.

  - `Float_t fADC`: The pedestal subtracted charge in units of ADC counts. The range of allowed charges is -15 to 240.75 in 1/4 ADC count units.
  - `Float_t fNoise`: The RMS of the pedestal (saved from the offline database) in ADC counts.
  - `Float_t fDNoise`: The RMS of the difference between pedestals of neighboring strips.
  - `Int_t fStatus`: 1=good strip, 0=bad strip according to offline database.
  - `Int_t fOBSP[3]`: The index into the `TObspBlock` (see Section **??**) of up to three GEANT particles which contributed to the charge on this strip.
  - `UShort_t fDigiCode`: The integer used as a datamember for the `TStnSiDigiCode` of the half ladder that this strip lives in (see Section 13.1).
  - Optional data members
    * `bool fStreamGeometryInfo`: If this variable has been set true while making the `TSiStripBlock` (see Section 13.4), then the following information is streamed out for each strip. This is useful for debugging purposes and to easily make plots of the positions of individual strips before clustering.
    * `TVector3 fGlobal`: Global coordinates of this strip.
    * `Float_t fLocal`: Local coordinates. The dead-center of the half ladder is the origin of this coordinate system.

13.4 TSiStripBlock

This is a collection of all silicon strips in the event. The `TStnSiStrip`'s (see Section 13.3) are stored in the flat `TClonesArray` in blocks of the same digicode (half ladder). If you want fast access to all strips in a given half ladder, you should call `TSiStripBlock::InitEvent()` to initialize the lookup tables. Then you can use `fIndexFirstHit[]` and `fIndexLastHit[]` to loop over all the strips on a half ladder.

In addition to the strips in the `TClonesArray`, the backend state and time since the previous L1 accept for each half ladder is also stored. There is also the option to turn on the streaming of the detailed geometry information of every strip (global and local coordinates). This makes the ntuple quite large, but can be useful for debugging and making plots. To turn this feature on, use the following talk-to in `StntupleMaker`: `makeSiStrips set 2`.

When filling this block, be sure that the `StorableRun2SiStripSet` is puffed in the `PuffModule`, or that you are rerunning clustering yourself. Otherwise, there will be no strips in the event record and this branch in your ntuple will be empty.

- Data members
  - `Int_t fNSiStrips`: Number of `TStnSiStrip`'s in the block.
  - `TClonesArray* fSiStripList`: Array of all strips.
  - `Int_t *fIndexFirstHit`: Array used as a lookup table for the index into the `TClonesArray` for the first strip on this ladder. If this half ladder has no strips, then the index is -1.

- `Int_t *fIndexLastHit`: Like previous array, but gives the index of the last hit on the half ladder. If this half ladder has no strips, then the index is -2.
- `Int_t fNDigiCodes`: Number of half ladders with strips.
- `Short_t *fBEState`: Backend state of this half ladder.
- `Short_t *fDtL1A`: Time since previous L1 accept for this half ladder.
- The other data members are streamed out, but are not generally accessed by the user. Instead they get puffed up into the lookup tables above by a call to `InitEvent()`.

• Member functions

- `Int_t InitEvent()`: Puffs up lookup tables for this event. If you aren't going to use the lookup tables, don't call this and it will save you some time.
- `TStnSiStrip* SiStrip(int i)`: Access to the strip in `fSiStripList` with index `i`.
- `Int_t FindStrip(int digi, int stripnum)`: Returns the index of the strip in `fSiStripList` with a given digicode and strip number. This is used primarily when filling the ntuple.

13.5 TStnSiHit

This class contains all the relavent information about a silicon cluster (also called a hit). It is a pared down version of the CDF class `TrackingObjects/SiData/SiHit`. For a link to the `TStnSiStrip`'s that make up this hit, see the link block in `TSvxDataBlock` in Section 13.6.

• Data members

- `TVector3 fGlobal`: Position of the hit in global coordinates. Since these are two-dimensional hits, the third coordinate is taken from the center of the half-ladder. In the case of hits that were used in a track, the global position is corrected for wafer level alignments based on the 3D track intersection with the wafer and the third dimension is taken from this intersection point.
- `Float_t fStripNum`: Position of hit on half ladder in units of strip number.
- `Float_t fLocal`: Same as previous local coordinate, but in units of centimeters. The origin is the center of the half ladder.
- `Float_t fQtotal`: Total (pedestal subtracted) charge of cluster in ADC counts.
- `Float_t fNoise`: The sum in quadriture of the noise of the strips making up the cluster.
- `Char_t fStatus`: Bit field. Bit 0=has bad strips, bit 1=neighbors are bad. Use the member functions to test if this is a good strip.
- `TStnSiDigiCode fDigiCode`: Digicode of the ladder that this hit lives on.
- `Int_t fUniqueID`: Reuse this data member of `TObject` to store information on which Monte Carlo particles contributed to the charge of the hit. Use member functions to get at this.

- Member functions
  - `Int_t NObsp()`: Number of Monte Carlo particles that produced this hit.
  - `Int_t Obsp(int i)`: Index into the `TObspBlock` (see Section ??) of the i-th particle contributing to this hit.
  - `Bool_t Good()`: This is a good hit.
  - `Bool_t HasBadNeighbor()`: This cluster contains a strip that is next to a bad strip and should be considered suspect.

13.6 TSvxDataBlock

This branch contains an array of all the silicon hits (`TStnSiHit`'s, see Section 13.5) in the event. It also has a link block to access all the strips in the `TSiStripBlock` (see Section 13.4) that contribute to a given hit. There are optional lookup tables which are puffed with a call to `TSvxDataBlock::InitEvent()` which enable the user to have easy access to all hits in a given half ladder (digicode).

There are also links from the hits to tracks, but this information is stored in a different block (see `TStnTrackLinkBlock` in Section 13.11).

- Data members
  - `Int_t fNSiHits`: Total number of `TStnSiHit`'s in this event.
  - `TClonesArray* fSiHitList`: Array of hits stored in blocks of the same digicode.
  - `Int_t* fIndexFirstHit`: Lookup table of index of first hit in a given digicode. This is created by a call to `InitEvent()`. If there is no hit in the given digicode, the index is -1.
  - `Int_t* fIndexLastHit`: Same as previous lookup table, but has the index of the last hit on the half ladder. Index is -2 if no hits on this digicode.
  - `TStnLinkBlock fSiStripLinkHit`: Link block of indexes into the `TSiStripBlock` which allows the user to have access to all strips used to create a given hit. See the example in `ana/TSiExample.cc` for a use case.
- Member functions
  - `Int_t InitEvent()`: Puffs up lookup tables for this event. If you aren't going to use the lookup tables, don't call this and it will save you some time.
  - `TStnSiHit* SiHit(int i)`: Access to the hit with index `i` in the array `fSiHitList`.
  - `Int_t IndexFirstHit()`: There are two forms of this function. One if you want to get the index of the first hit given a barrel, ladder segment, phi wedge, layer and side, and the other if you know the digicode already. This needs to have `InitEvent()` called beforehand.
  - `Int_t IndexLastHit()`: Same as previous function, but returns the index of the last hit on the half ladder.

13.7 `TStnSiIsect`

This object contains some information about a track intersection with a particular silicon half ladder. It can be useful for making residuals with silicon hits.

These objects are stored in the array in the `TSiIsectBlock` of Section 13.8. If you are looping over tracks in the track block, then you can use the links in `TStnTrackLinkBlock` (Section 13.11) to access these intersections.

- Data members
  - `TStnSiDigiCode fDigiCode`: The digicode of the half ladder that this track intersection is coming from.
  - `TVector3 fGlobal`: Global coordinates (corrected for wafer-level alignment) of track intersection with this half ladder.
  - `Float_t fStripNumPhi`: Intersection point in strip units on the axial side.
  - `Float_t fStripNumZ`: Intersection point in strip units on the stereo side.
  - `Float_t fLocY` and `Float_t fLocZ`: Same as previoius local coordinates, but in centimeters.
  - `TBitset fActiveRegion`:

Bit	If set, then passed thru active
0	phi area of axial side
1	z area of axial side
2	z area of stereo side
3	phi area of stereo side

13.8 `TSiIsectBlock`

This branch contains a collection of the intersections of tracks with silicon halfladders. If you are looping over tracks, then you can use the correspondance object, `TStnTrackLinkBlock` (see Section 13.11), to get access to all intersections for a given track. You also have immediate access to all silicon hits for this track for comparison using that link block.

- Data members
  - `Int_t fNSiIsects`: Total number of track-silicon intersections in this event.
  - `TClonesArray* fSiIsectList`: Array of `TStnSiIsect`'s.
- Member functions
  - `TStnSiIsect* SiIsect(int i)`: Access to the i-th intersection in the array `fSiIsectList`.

13.9 `TStnSiGeantIsect`

Similar to the track intersection object, `TStnSiIsect`, this object contains information of Monte Carlo particle intersections with silicon half ladders. It stores where the particle went and how much energy it lost while traversing the half ladder.

This object is a pared down version of the CDF class `PropagatedSiParticle`.

- Data members
  - `TStnSiDigiCode fDigiCode`: The digicode of the half ladder that this Monte Carlo particle traversed.
  - `Int_t fUniqueID`: Reuse this data member from `TObject` to store the index into the `TObspBlock` of the particle that hit this ladder.
  - `TLorentzVector fEntryMomentum`: 4-momentum of the particle at the point of entrance to the half ladder.
  - `TLorentzVector fExitMomentum`: 4-momentum upon exit.
  - `TVector3 fEntry` and `fExit`: The entry and exit points in global coordinates.
  - `TBitset fActiveRegion`:

Bit	If set, then passed thru active
0	Entrance in Phi area of side 0
1	Entrance in Z area of side 0
2	Entrance in Z area of side 1
4	Entrance in Phi area of side 1
5	Exit in Phi area of side 0
6	Exit in Z area of side 0
7	Exit in Z area of side 1
8	Exit in Phi area of side 1

  - `Float_t fDE`: Total energy loss in GeV in this half-ladder.
  - `Float_t fRadLen`: Fraction of radiation length traversed.
  - `Float_t fDistToNearPhi`: Distance from (exit-entrance)/2 to center of nearest strip. The Hall Effect should be visible in charge deposition models that include it.
  - `Float_t fDistToNearZ`: Same as previous, but for strips on stereo side.
  - `Int_t fInitPhi`: Initial hit phi strip.
  - `Int_t fFinalPhi`: Final hit phi strip.
  - `Int_t fInitZ` and `fFinalZ`: Same as previous, but stereo side.

13.10 `TSiGeantIsectBlock`

This branch contains all the intersections of Monte Carlo particles (`TStnSiGeantIsect` of Section 13.9) with the silicon detector elements. It also has a link block to allow the user to have access to the particle in the `TObspBlock` that made the itersection.

- Data members
  - `Int_t fNSiGeantIsect`: Total number of `TStnSiGeantIsect`'s in this event.
  - `TClonesArray* fSiGeantIsectList`: Array of intersections.
  - `TStnLinkBlock fIsectLinkObsp`: Link block from OBSP index to the number of intersections that it made. This is useful if you are looping over particles in the `TObspBlock` and you want access to the intersections in `fSiGeantIsectList`. This link block is puffed by a call to `TSiGeantIsectBlock::InitEvent()`.

- Member functions:
  - `Int_t InitEvent()`: Puffs the link block `fIsectLinkObsp` for this event.
  - `TStnSiGeantIsect* SiGeantIsect(int i)`: Access to the i-th intersection.

13.11 TStnTrackLinkBlock

This is a link block which stores the correspondence between tracks and their silicon hits and intersections with the silicon detector. This branch is especially useful if you are looping over tracks in the track block, `TStnTrackBlock`, and you want access to the above information.

- Data members
  - `TStnLinkBlock fSiHitLinkTrk`: Links from tracks to silicon hits in the `TSvxDataBlock` (see Section 13.6).
  - `TStnLinkBlock fSiIsectLinkTrk`: Links from tracks to the track intersections in the `TSiIsectBlock` (see Section 13.8).